Implementation of the Potential Field Method for motion planning at the TURTLEs

J.C.T. (Jan) Romme - 0755197
N.A. (Niek) Wolma - 0633710

Coaches:
Dr.ir. G.J.L. Naus
S.A.M. Coenen

Supervisor:
Dr.ir. M.J.G. van de Molengraft

Eindhoven University of Technology
Department of Mechanical Engineering
Control Systems Technology

Eindhoven, August, 2013
Abstract

This paper is a description of the implementation process of the Potential Field Method on the TURTLE soccer robots used by TECHUNITED. The PFM is a method in which a robot can drive to a target without a preprogrammed path. It is capable of avoiding obstacles placed at random positions. The steps taken during this project will help the TECHUNITED team decide whether or not to choose the PFM as their future motion planning system.

This project has taken the PFM from simulations in MATLAB to the actual TURTLEs. The basic PFM has been converted into C-code and has been tested on both the simulator and in the field on a TURTLE. Furthermore a few additions have been made to the basic PFM, namely a function forcing the TURTLEs to stay within field lines and a function enabling a TURTLE to escape local minima.

The effects of incorporating relative velocities have been evaluated, foremost resulting in better obstacle avoidance. Upon implementing the PFM, parameter tuning again proved to be critical. Imbalances in underlying components of the algorithms result in collisions and other undesired behaviour like entrapment in local minima. Tuning the PFM will be the next step in taking ”motion planning using the PFM” to ”playing actual soccer using the PFM”.

All together the PFM is deemed capable of motion planning for the TECHUNITED soccer team, and a follow up on this project is recommended.
A.2.1 Results ......................................................... 28
A.3 Distance between camera and object ........................................ 29
A.3.1 Results ......................................................... 30

B Installing the PFM on your DEV-pc ........................................... 32
C PFM in C-code ....................................................... 34
D Original Assignment ....................................................... 51
1 Introduction

RoboCup is an international research and education initiative focusing on artificial intelligence (AI) and intelligent robotics. To stimulate participation, a competition is set up, including rescue leagues, service robotic leagues and soccer leagues in different forms. The highly dynamic environment of soccer is a perfect platform for application of AI and intelligent robotics and, as such, leads to innovations that benefit society [1]. Tech United Eindhoven participates in three leagues: the @Home, the Humanoid, and the Middle-Size League [2]. In the Middle-Size League, a team of five robots (called TURTLEs) play soccer autonomously. These wheeled robots are 80 cm in height, weigh 35 kg, reach a top speed up to 5 m/s and shoot a ball with a speed of 10 m/s. The robots are required to have all sensors on-board. The fast-paced soccer game of this league poses great challenges in several fields such as mechatronics, software, strategy, cooperation and AI [2].

After a literature study by Wilschut [4] and the first implementation steps by van der Poel [5], this project is the next step in implementing the Potential Field Method for the TURTLE soccer robots. The assignment can be found in appendix D.

1.1 Problem definition

An important part of the TURTLE control software involves motion planning. Motion planning includes high-level decisions on, e.g., where to position on the field or via what side to attack, and low-level computations on how to avoid collisions with opponents and on the exact trajectory to drive. Given the motion planner currently used by the TURTLEs, various points for improvement have been identified. One important example involves taking into account velocity estimation and prediction of opponents. This is, however, difficult or even impossible to do using the currently implemented planner. Based on a literature study by Coenen [3], the potential field method (PFM) has been identified as a suitable method including the proposed improvements. Additional research and implementation are required to evaluate this method in practice.

1.2 Action plan

Currently the PFM has been evaluated in a Matlab simulation environment by Wilschut [4], after which van der Poel [5] started implementing the PFM into the TURTLE software which is written in C within a Matlab/Simulink environment.

This project starts by completing the basic PFM in TURTLE code. This is done by incorporating opponent velocity, completing the repulsive force component and completing the sliding mode control.

Furthermore two functionalities will be added to adapt the basic PFM to specific robot soccer situations, being a field line obstacle functionality to keep the robot inside the playing field and a escape force functionality to escape local minima. Finally the current obstacle speed estimation will be evaluated.
1.3 Potential Field Method

The PFM motion planning is done iteratively, meaning that for each step the PFM determines the optimal direction of motion, and is thus better suited for dealing with highly dynamic situations. In determining the desired direction of motion, the PFM incorporates both the TURTLE and obstacle velocities. When executing motion using the PFM, the robot is attracted by the target and repelled by obstacles as can be seen in Figure 1.1.

To realize this, the space in which the robot has to navigate is treated as a potential field. The target position for the robot is considered a global minimum and obstacles are considered global maxima. By differentiating this field an artificial force $\vec{F}$ is obtained. This force is the sum of all attractive $\vec{F}_{\text{att}}$ and repulsive $\vec{F}_{\text{rep}}$ forces generated by the field. The netto force $\vec{F}$ is then applied to the robot’s inertia to calculate the desired acceleration. In depth information about the PFM (algorithms) can be found in Wilschut [4].

A guide to install the PFM on your DEV-pc can be found in appendix B. The entire C-code file can be found in appendix C.


2 Field Line Obstacle Functionality

2.1 Theory

The robot has to perform within a limited area, i.e. staying within the field lines of the soccer field. To accomplish this behaviour a virtual obstacle called a 'field line obstacle' is placed on the field line as described in Mouton [6]. This virtual obstacle is placed in the direction of motion of the robot at the point where it would cross the field line. This solution is illustrated in Figure 2.1.

\[ \vec{p}_{IP} = \vec{p}_{R} + \lambda \vec{n}_{VR} \] (2.1)

In vector form this relationship is described by equation 2.2.

\[ \begin{pmatrix} x_{IP} \\ y_{IP} \end{pmatrix} = \begin{pmatrix} x_{R} \\ y_{R} \end{pmatrix} + \lambda \begin{pmatrix} n_{x,VR} \\ n_{y,VR} \end{pmatrix} \] (2.2)

For the two horizontal field lines \( x_{IP} \) is known and for the two vertical field lines \( y_{IP} \) is known. The corresponding \( \lambda \) for a vertical line is given by (2.3).

\[ \lambda = \frac{y_{IP} - y_{R}}{n_{y,VR}} \] (2.3)

The corresponding \( x_{IP} \) is given by (2.4).

\[ x_{IP} = x_{R} + \lambda n_{x,R} \] (2.4)
Finding the coordinates for a horizontal field line is done the same way. This method generates 4 intersection points, 1 for each field line. The correct intersection point is determined by evaluating the $\lambda$ for each intersection point, the smallest positive $\lambda$ corresponds with the relevant intersection point.

The field line obstacle has different parameters as a normal obstacle to allow for tuning. These parameters are a 'field line offset' to put the field line obstacle behind the line, and a 'field line obstacle influence radius' to alter the influence radius of the field line obstacle. This field line obstacle has no radius, since a physical collision with the robot is not possible. These parameters are illustrated in in Figure 2.2.

![Field line obstacle and the geometric parameters.](image)

Figure 2.2: Field line obstacle and the geometric parameters.

Furthermore a repulsive force scaling factor $\eta_{fo}$ is introduced to give the field obstacle a tunable repulsive force function.
2.3 Field Line Obstacle Functionality Experiments

The experiments as described in the sections below are conducted to check if the field line obstacle functionality works and how the field line functionality compares with the current motion planning.

2.3.1 Driving to target behind the field line

To test if the field line functionality works the robot is given a target behind the field line. This situation is illustrated in Figure 2.3.

As the robot is driving to the target a field line obstacle is placed behind the field line. When the robot is within the influence radius the field line obstacle exerts a repulsive force on the robot causing the robot to slow down until the robot is stationary and keeping the robot within the applied field boundaries. By decreasing the repulsive scaling factor of the field line obstacle, the robot will approach the field line obstacle at higher speeds but simultaneously the chance of the robot crossing the field line will increase. However the field line functionality works.
2.3.2 Driving past an obstacle near the field line while having the ball

The robot is driving next to and parallel to the field line having the ball. An obstacle is standing close to field line and the robot has to pass this obstacle. This situation is illustrated in Figure 2.4.

![Figure 2.4: Robot has to pass obstacle near the field line.](image)

The current motion planning has two solutions to this problem. If the robot is at great distance from the obstacles it defines a subtarget above the row of obstacles. However when the robot is close to the lowest obstacle in the row the robot passes between the obstacles and the field line in a way that the ball will be held within the field or on the field line. This way it is capable of dealing with an extreme case of this situation where the distance between the edge of the obstacle and the field line is having the same magnitude as the ball radius. This situation and the solution are illustrated in Figure 2.5.

![Figure 2.5: Current motion planning solution to driving past an obstacle near the field line having the ball.](image)

The PFM however places a field line obstacle on the field line as the direction of motion of the robot changes. When the situation as illustrated in Figure 2.5 is applied with the PFM, the robot gets trapped into a local minimum because the field line obstacle exerts a repulsive force onto the robot as illustrated in Figure 2.6. To escape this local minimum an escape force is applied which causes the robot to travel a longer path as the current motion planning. Local minima are evaluated in the next chapter. With the situation as illustrated in Figure 2.4 the field line obstacle will slow down the robot, as the robot approaches the lowest obstacle the direction of motion is pointing to the field line below the obstacle and the field line obstacle which is placed there exerts a repulsive force onto the robot causing the robot to slow down. As the robot continues to travel the direction of
motion starts pointing to the target and with that the field line obstacle has no influence on the robot anymore. The robot will thus pass the obstacles and reach the target.

### 2.3.3 Retrieve the ball on a field line

The robot has to retrieve the ball on a field line. When the ball is on the back-line of the robot’s own half the current motion planning approaches the ball directly, i.e. no additional subtargets are set which allows the robot to travel a shorter path, however overshoot causes the robot to drive the ball over the back-line. For the other field lines the robot puts the target onto the ball and plans an additional subtarget so the robot approaches the ball sideways to prevent the ball crossing the field line, however this approach is not always successful since again overshoot sometimes causes the robot to drive the ball over the fieldline. The PFM approaches the ball directly in all cases. As the robot approaches the ball a field line obstacle is placed behind the ball as illustrated in Figure 2.7. The field line obstacle exerts a repulsive force onto the robot. By correctly tuning the field line obstacle parameters the field line obstacle prevents overshoot and with that preventing the ball to get over the field line. However by overtuning the influence radius and repulsive scaling factor of the field line obstacle, the robot gets trapped into a local minimum, whichwithholds the robot from retrieving the ball.

### 2.4 Conclusion

The field line obstacle functionality proves to work. However passing obstacles near the field line having the ball is still a problem with this functionality because the robot gets trapped into a local minimum. With the escape force functionality explained in the next
chapter this problem can be overcome. Furthermore retrieving the ball from a field line proves to be safer with the PFM as with the current motion planning because the PFM prevents overshoot with the field line obstacle functionality keeping the ball within the allowed boundaries.
3 Escape Force Functionality for Local Minima

3.1 Theory

A solution for escaping local minima is described by Wilschut [4]. Local minima occur at null-potential areas, i.e. the sum of the attractive force and repulsive force is zero, so when equation 3.1 holds.

\[ ||\vec{F}_{total}|| = ||\vec{F}_{att} + \sum_{i=1}^{N_o} \vec{F}_{rep,i}|| = 0 \] (3.1)

A local minimum is identified when conditions (3.2) and (3.3) are true where \(a\) and \(b\) are arbitrary chosen constants. These conditions are explained by Wilschut [4].

\[ \frac{||\vec{F}_{total}||}{||\sum_{i=1}^{N_o} \vec{F}_{rep,i}||} < b \] (3.2)

\[ \cos(\angle \vec{F}_{att} - \angle \sum_{i=1}^{N_o} \vec{F}_{rep,i}) < -\cos(c) \] (3.3)

When a local minimum is identified, an escape force \(\vec{F}_e\) is introduced in the direction perpendicular to \(\vec{F}_{rep}\). This steers away the robot from the local minimum since \(\vec{F}_{total}\) is not pointing to the local minimum anymore causing the robot to go around the obstacle. The effect of the escape force is illustrated in Figure 3.1.

![Figure 3.1: Effect of the escape force.](image-url)
3.2 Implementation

The angle $\angle \vec{F}_{\text{att}} - \angle \sum_{i=1}^{N_o} \vec{F}_{\text{rep},i}$ as seen in (3.3) is calculated using the dot product. This angle is given by (3.4).

$$\angle \vec{F}_{\text{att}} - \angle \sum_{i=1}^{N_o} \vec{F}_{\text{rep},i} = \acos \left( \frac{\vec{F}_{\text{att}} \cdot \sum_{i=1}^{N_o} \vec{F}_{\text{rep},i}}{||\vec{F}_{\text{att}}|| \cdot ||\sum_{i=1}^{N_o} \vec{F}_{\text{rep},i}||} \right)$$ (3.4)

Conditions (3.2) and (3.3) can now be evaluated. One additional condition (3.5) is added to assure the escape force does not steer the robot away from the target position which is explained by Wilschut [4].

$$||\vec{p}_{\text{tar}} - \vec{p}_{\text{r}}|| > d$$ (3.5)

Where $d$ is the minimum distance between the robot and the target for the escape force to be applied. Once these three criteria are met the obstacle which is closest to the robot is used to calculate the escape force $\vec{F}_e$ is given by (3.6).

$$\vec{F}_e = \alpha_e \frac{1}{\left(\rho_s - \rho_r - \rho_m\right)^2} \vec{n}_{\text{rep}\perp}$$ (3.6)

Where $\vec{n}_{\text{rep}\perp}$ is a unit vector perpendicular to the repulsive force $\vec{F}_{\text{rep}}$ and $\alpha_e = 10^z$, $z = \log_{10}(\vec{F}_{\text{rep}})$. 

Before the escape force is added to the artificial force, the angle between the velocity vector $\vec{v}_{\text{R}}$ and the unit vector $\vec{n}_{\text{rep}\perp}$ is checked to be less or equal to $\frac{1}{2}\pi$. If not $\vec{n}_{\text{rep}\perp}$ is set to $-\vec{n}_{\text{rep}\perp}$. This prevents the robot from slowing down by the escape force and allows for a faster escape since the robot can continue in the direction it is already travelling.
3.3 Escape Force Functionality Experiments

The experiments as described in the sections below are conducted to find the possibilities and limitations of the escape force functionality for escaping local minima. The behaviour of the escape force functionality is compared with the current motion planning and with the basic PFM.

3.3.1 Drive into a straight wall of obstacles

The robot is driving into a straight wall of obstacles, the target is behind the obstacles. The situation is illustrated in Figure 3.2.

![Figure 3.2: Straight wall of obstacles.](image)

The current motion planning solves this situation by defining a subtarget above or under the obstacles as seen in Figure 3.3.

![Figure 3.3: Current motion planning defines a subtarget.](image)

When the PFM is applied without the escape force functionality the robot gets trapped into a local minimum because the repulsive force exerted by the obstacles overrides the attractive force of the target. With the escape force functionality the robot manages to escape the local minimum and reach the target. However the path length travelled is much longer since the robot does not drive in straight lines as the current motion planning. The travelled path with the escape force functionality is illustrated in Figure 3.4. To get the path more fluently the escape force needs to start earlier. The escape force is only applied when a repulsive force is working on the robot under the condition that a local minima is identified. To achieve that the repulsive force is working earlier the influence radius is increased. Secondly the repulsive scaling factor $\eta$ is tuned down to prevent the robot from losing all its speed. The new travelled path is illustrated in Figure 3.5.
3.3.2 Driving between two obstacles

The shortest path of the robot to the target is driving between two obstacles as illustrated in Figure 3.6.

The current motion planning takes the shortest path at full speed as long as the robot fits between the gap caused by the two obstacles. When the gap is too small the current motion planning defines a subtarget above or below the obstacles. The PFM without the escape force functionality gets trapped into a local minimum. Using the escape force functionality the robot manages to travel the shortest path. However the speed of the robot with the PFM is much lower because as the robot approaches the obstacles the repulsive force exerted by the obstacles causes the robot to slow down. To increase the speed of the robot the repulsive scaling factor $\eta$ is tuned down. However as $\eta$ gets lowered the chances of collision increase since the obstacles repulsive force has less effect on the attractive force. Alternatively by increasing the influence radius and tuning down $\eta$ the robot will drive above or below the obstacles without losing all of its speed.
3.3.3 Drive into a C-shaped lined up formation of obstacles

The robot is driving into a C-shaped lined up formation of obstacles. The target is behind the C-shape. The situation is illustrated in Figure 3.7.

![Figure 3.7: C-shape formation of obstacles.](image)

The current motion planning defines a subtarget above or below the obstacles and drives in two straight lines to the target. The PFM without the escape force functionality gets the robot trapped into a local minimum. When the escape force functionality is applied the robot escapes from the local minimum, however as the robot goes upward (or downward) the repulsive force exerted by the last obstacle slows down the robot’s speed to zero. Then the attractive force exerted by the target causes the robot to go downwards. This process repeats at the bottom causing the robot to oscillate and prevents the robot from reaching the target. To prevent the robot from getting 'into' the C-shape the escape force has to start earlier. Again the influence radius has to increase and the repulsive scaling factor has to decrease. However the chances of collision with the top or lowest obstacle increase since the repulsive force has less effect on the attractive force.

3.4 Conclusion

The escape force functionality is able to escape local minima. However for different situations the escape force needs different parameters to travel collision free and to travel as fast as possible. A better approach would be a system which is capable of discerning different scenarios and able to select the optimal set of parameters from a library.
4 Effect of incorporating relative velocity on motion planning

Incorporating relative velocity into motion planning changes the TURTLE’s behaviour in certain field situations. In order to demonstrate this a few of these field situations are evaluated in this chapter.

In comparison to the current motion planning the PFM incorporates three more variables into determining the desired direction of motion; being object speed, target speed and the robot’s own speed. These are then used to calculate the relative speeds between robot and the target and between the robot and nearby obstacles. In turn these are used by the new attracting and repelling algorithms.

4.1 Relative velocity between robot and obstacle

This section presents two field situations in which the incorporation of the relative velocity would differ from the current motion planning.

Situation 1: Attempted enemy block

The robot tries to reach a static target position, while an enemy object will attempt to block the robot’s path as shown in figure 4.1. The red dashed line signifies the centre axis in the situation.

To the current motion planner the right side of the field would seem open, and thus a path would be planned on the right side of the red centre line. This would result in either a successful block from the enemy, a collision or a long detour to avoid this collision.

The PFM on the other hand will plot the path on the left side of the centre line. This path
may seem blocked in figure 4.1, but by incorporating the relative velocity the PFM can predict that the left half will soon be open and the right half will soon be blocked, making a path on the left half the preferred choice.

**Situation 2: Scrum**
The robot and an enemy object are facing each other with the target (exactly) between them, as shown in figure 4.2.

![Figure 4.2: Scrum.](image)

The current motion planner sees no obstacles directly between the robot and the target and will plan a direct path accordingly.
The PFM will do the same, however when approaching the target the robot will be repelled by the obstacle, preventing the PFM from reaching the target. In this situation the standard PFM seems less apt than the current motion planning and is likely to require some specific tuning. This situation is very similar to the problem defined in chapter 2, the field line problem.

### 4.2 Relative velocity between robot and target

By incorporating the relative velocity between the robot and the target the PFM will steer towards an intercept point. The current motion planning also uses interception points, however these are determined on a higher strategy level.

**Situation 1: Intercept moving obstacle**
A (stationary) robot has to intercept a moving obstacle, as shown in figure 4.3
The current motion planner is capable of navigating towards a set of coordinates, and so does not incorporate target speed. This is solved by calculating an intercept point and sending the robot there.

The PFM requires the current target position and speed to determine the desired direction of motion. Even though the methods differ slightly, the results are very similar.

4.3 Implementation

Continuing on Wilschut [4] and van der Poel [5], getting the TURTLEs to plan motion using PFM is rather straightforward. The basic implementation however does not drive collision free.

Problems occur when one or more of the components of either the attractive or repulsive algorithm become too large. In these cases they displace the other components, resulting in an unbalanced net virtual force.

E.g. Wilschut [4] suggests a quadratic distance/attraction function. Suppose a TURTLE has to reach a target at great distance. Since the attractive force is a function of the distance between the robot and the target, the attractive force will also be great. When blocked by an obstacle, the quadratic attractive function will be far greater than the repulsive function. The net force will mainly be affected by the attractive component, making the TURTLE collide with the obstacle. This specific situation is solved by changing the attractive function into a root function, preventing the attractive force becoming disproportionately large.

In order to balance/tune the individual components for different field situations, the function tester could be a useful tool. In the function tester static situations can be analysed and reproduced. At the same time the artificial forces can be visualized. Figure 4.4 shows a demonstration of the attractive component.
The green vector visualises the first part of the attractive algorithm, pointing towards the target at all times (the orange ball). The grey and orange vectors visualize the speeds of the robot and the ball, yielding the second part of the attractive algorithm denoted by the blue vector. The blue and green vector together form the yellow vector, the net attractive force. The net attractive force is pointed towards an intercept point.

The function tester can be used to analyse several field situations including opponents. In order to play soccer using the PFM several situations have to be tuned.

4.4 Conclusion

The difference between the 2 path planning methods is most evident for obstacle avoidance. In this case the PFM is more extensive than the current system. The added functionality of incorporating opponent speeds has the potential to plan more efficient paths and avoid more collisions.

When attempting to reach a moving target the current motion planner and the PFM use very similar methods, resulting in similar results. The actual implementation of the PFM for game purposes presents more of a challenge, since good tuning is key to efficient performance. The function tester, a tool used to analyse static field situations, may be useful in determining the right parameter settings.
5 Velocity Estimation Analysis

Since the PFM takes the relative velocity between robot and obstacle and/or between robot and target into account during motion planning, these velocity estimations must be accurate. This chapter analyses the quality of these estimations.

The relative velocity is calculated by subtracting the robot’s velocity from the target/obstacle velocity (henceforth referred to as object velocity). Eventual errors in the relative velocity estimation are thus caused by errors in the robot’s own velocity estimation and/or errors in the object/target velocity estimation.

For this first evaluation potential errors in the robot’s own velocity estimation are deemed insignificant compared to potential errors in object velocities. The robot’s own velocity is determined using the encoders. Object velocities on the other hand are determined by the vision system. Because this method contains a great number of variables that could influence accuracy, these velocity estimations are expected to be significantly less accurate. This purpose of this chapter is twofold; presenting experiments that determine the accuracy of object velocity estimations and an evaluation of significant errors found during the experiment.

5.1 Object velocity analysis

To determine the accuracy of object speed analysis, the following variables that could influence results have been identified.

I Speed of the object

II Positioning/orientation of the camera with respect to the object path

III Distance between camera and object path

IV Number of cameras

First it is assumed that when determining object speed using multiple cameras, the velocity estimation improves since sensor fusion occurs. Looking for the worst-case scenario, all experiments will be conducted using a single camera.

Second it is assumed that the three remaining variables do not influence each other. Therefore the influence of these four variables can be tested using three different experiments, which are described and discussed in A.

5.2 Results

Summarizing appendix A:

Trajectory: When object speeds are very low the system is unable to accurately determine the direction of motion. At higher speeds perceived direction of motion is accurate.

Absolute speed: For object speeds up to 2.5 m/s the system is able to give good estimations (deviations < 0.1 m/s). The camera was unable to detect speeds of 3.5 m/s.
Noise: As mentioned before, stationary objects are not always perceived as stationary. The amount of noise varies per experiment, from 0 to errors of almost 0.5 m/s. Furthermore, the system sometimes loses track of the object for a short amount of time (≈ 0.3s), during which the perceived speed drops to 0.

5.2.1 Error analysis

Trajectory: When the object is stationary the system (sometimes) perceives small speed vectors pointing in random directions. The influence of these disturbances on the desired direction of motion is very small, since the magnitude of these vectors is very small. However, a simple “if-statement” in the code, for example ignoring all object speeds below 0.1 m/s, would assure that these disturbances have no negative influence.

Absolute Speed: The magnitude of the velocity estimation is accurate up to 2.5 m/s, and therefore requires no further action. A single robot was unable to discern speeds greater than 2.5 m/s, caused by the object being out of vision range due to a faulty test setup. This can be solved by adjusting the experiment.

Noise: As mentioned before the noise from stationary objects can easily be filtered out. The noise on moving objects is minimal and requires no further action. Furthermore, the perceived drops to 0 m/s temporarily negate the relative velocity component. Even though this is not optimal, the PFM is still functional without the relative components. Furthermore, these drops are of very short duration and can therefore be ignored.

5.3 Conclusion

Stationary objects generate some noise which can be easily filtered out, thus causing no significant errors. Object speeds up to 2.5 m/s are perceived accurately. For a conclusion on speeds above 2.5 m/s a new experiment has to be conducted. However, for speeds below 2.5 m/s the velocity estimation is accurate enough for use in the PFM.
6 Conclusion

This project has taken the PFM from simulations in MATLAB to the actual TURTLEs. The basic PFM has been converted into C-code and has been tested on both the simulator and in the field on a TURTLE. Furthermore a few additions have been made to the basic PFM, namely a function forcing the TURTLEs to stay within field lines and a function enabling a TURTLE to escape local minima.

The robot is capable of staying within the field line boundaries using a virtual field line obstacle. This field line obstacle is placed in the direction of motion of the robot. A benefit from this functionality is that when the robot has to retrieve a ball on the field line it does not overshoot like the current TURTLE’s motion planning and thereby preventing the ball from crossing the field line. However without the escape force functionality the robot can get trapped in a local minimum between a real obstacle and a virtual obstacle.

The escape force functionality is a function to escape local minima. The escape force functionality consists of a local minimum detection algorithm and applies a virtual escape force which pushes the robot out of the local minimum. In certain situations with local minimum the robot is slowed down to a near stop. To prevent this behaviour the parameters have to be changed, however changing the parameters may cause collision in other situations. Therefore the parameter tuning should be situation dependent.

The effects of incorporating relative velocities have been evaluated, foremost resulting in better obstacle avoidance. Upon implementing the PFM, parameter tuning again proved to be critical. Imbalances in underlying components of the algorithms result in collisions and other undesired behaviour like entrapment in local minima. Tuning the PFM will be the next step in taking ”motion planning using the PFM” to ”playing actual soccer using the PFM”.

Since object velocity estimations are being used by the PFM, the accuracy of these estimations has to be analysed. When dealing with stationary objects the robot perceives the object to have a small velocity in random directions. This is caused by noise. However the contribution of the faulty signals are not significant and therefore ignored. Therefore the robot’s velocity estimation of objects up to 2.5 m/s is accurate. The velocity estimation of velocities above 2.5 m/s should be re-evaluated in a new experiment.

All together the PFM is deemed capable of motion planning for the TECHUNITED soccer team, and a follow up on this project is recommended.

6.1 Recommendations for future work

Parameter tuning: The performance of the PFM is greatly affected by the chosen parameters. Currently the PFM works with a set of parameters that is capable of avoiding
collisions and reaching a target. However many field situations require a different set of parameters in order to reduce travel time. e.g. the PFM is currently able to escape local minima but this takes relatively long. Critical situations should be identified and the parameters should be tuned accordingly. The robot’s themselves should ideally be able to identify what parameters are required to achieve the desired outcome.

**Narrow corridor instability:** Both Wilschut [4] and van der Poel [5] have mentioned unstable behaviour in narrow corridors. Even though this has not occurred during this project, potential problems within narrow corridors should be investigated.

**Application of the PFM to actual robot soccer:** Currently the work has been focused to get the motion planning working based on the PFM. The next step should be to focus playing soccer with the PFM instead of just driving to a target. This requires integration with high level strategy.

**Integration into software architecture:** The current implementation of the PFM is done by terminating and replacing several signals in the SIMULINK environment, creating a messy and unclear system.
References


A Object velocity analysis

As mentioned in the report the accuracy of the velocity estimation must be reviewed using the following variables:

I Speed of the object
II Position of the camera with respect to the object
III Distance between camera and object

For each variable the used experiment will be described, along with the results.

A.1 Speed of the object

To determine the influence of object speed on measurement accuracy, the experiment as shown in figure A.1 was carried out.

![Figure A.1](image)

Figure A.1: Experiment used in determining influence of object speed on measurement accuracy.

The robot, illustrated by the black dot, is stationary whereas the object (another robot) moved around the red pattern in a clockwise motion. Whenever the object was out of vision range (determined using greenfield) the object’s speed was increased using the tunable parameters. In this setting the object completed the loop four times at four different speeds, being 0.5, 1.5, 2.5 and 3.5 m/s.

A.1.1 Results
The plot as shown in figure A.2 is the result of a simout block in the worldmodel on the stationary robot. The red vx and blue vy line clearly show 4 object passings, first approaching the robot resulting in negative values and then moving away from the robot resulting in positive values. The approach path of the object is at an angle compared to the Y-axis, as shown by the combined vx and vy values. When moving away the obstacle follows the Y-axis, solely generating vy values.

Combining vx and vy using the Pythagorean theorem gives us an absolute speed, plotted in the green dash-dot line. The angle at which the obstacle moves is calculated using the arctan and represented by the dotted magenta line.

**First pass, 0.5 m/s:** The object moves very slowly at constant speed, as shown by the plot. The measured speed (vabs) levels at the set 0.5 m/s and dips to 0 when the robot changes direction of motion.

**Second and third pass, 1.5 m/s and 2.5 m/s:** Even though vabs varies slightly more from the set speed than at the first pass, the robot still accurately observes the objects velocity. In the approaching part of the third pass the object barely reached full speed, as seen by the fact that the vx and vy values peak instead of plateau.

**Fourth pass, 3.5 m/s:** Here the measurement doesn’t reach the set 3.5 m/s. As observed
by the third pass during approach, the object doesn’t have enough wavelength to fully accelerate to the set speed. When moving away the object has plenty of wavelength but the object is most likely out of vision range before the set speed is reached. Furthermore note the data loss at \( t \approx 66 \text{s} \), where the observed speed drops to 0 for a few milliseconds. These moments of data loss could possibly be caused by a temporary localisation problem or a bad image from the vision system. These moments of data loss occur regularly, but only last for milliseconds at a time and are therefore not considered problematic.

**Angle:** The detected angle of approach is remarkably constant for every pass. When target speeds get close to 0 the detected angle of approach spikes. Even though the error is relatively big, it occurs when \( v_{\text{abs}} \) is close to zero. When \( v_{\text{abs}} \) is near 0, the forces generated become significant.

### A.2 Position of the camera with respect to the object

To determine the effect of camera placement w.r.t. the object the experiment as shown in figure A.3 was carried out.

![Figure A.3: Experiment used in determining influence of camera position on measurement accuracy.](image)

The robot, illustrated by the black dot, is stationary whereas the object moves over the red line. The robot is then moved to create the second setup and the object travels along the same path at the same speed (2.5 m/s). Note that in case of a perfect velocity estimation the speed plots from both experiments should be identical.

### A.2.1 Results
Figure A.4: Measured speeds of object.

**Robot in line with object path:** The measured speed does reach the set 2.5 m/s. Also, the measured path angle is approximately 0. The robot did however not capture the object’s full path because of the limited vision range.

**Robot perpendicular to object path:** Apart from the data loss at $t = 4.5s$, this experiment has generated a very smooth velocity profile in which acceleration and deceleration are clearly shown. In this setting the robot was able to capture the object’s path completely. Note that during the ‘in line’ measurement object speed 0 is actually recorded as speed 0. When the distance between robot and object become greater, like in the perpendicular setup, small disturbances occur resulting in spikes in the angle.

### A.3 Distance between camera and object

To determine the influence of the distance between the robot and the object the experiment as shown in figure A.5 was carried out.
The object moves over the red line at a constant speed of 2.5 m/s while the robot is placed at various distances $d$ from the object.

A.3.1 Results

Figure A.6: Measured speeds of object.
\textbf{d1 and d2, 1 m and 2 m:} The robot records the set speed of 2.5 m/s at both distances. Note the disturbances at the beginning of both measurements. The cause of these disturbances is unknown.

\textbf{d3, 3 m:} The robot barely registers the passing obstacle.
B Installing the PFM on your DEV-pc

1. Check out the DevPC to the latest version.
   - Launch SmartSVN
   - Update to HEAD

2. Copy `makeASSERTion_PFM.m` to the motion folder
   - `/home/robocup/svn/trunk/src/Turtle2/Motion/`

3. Copy `determinedirection.c`
   - `/home/robocup/svn/trunk/src/Turtle2/Motion/src/`

4. Start MatLab
   - Type into the command window: `bus_manager`
   - Select: `tunable_parData_motion_bus`
   - Add the following variables to the bus using add variable (case sensitive):
     - `PFM_m = 1.75`
     - `PFM_n = 1.75`
     - `PFM_alpha_p = 1.0`
     - `PFM_alpha_v = 1.0`
     - `PFM_rho_O = 0.25`
     - `PFM_ether = 10`
     - `PFM_ether_wall = 5.0`
     - `PFM_ether_wall = 2.5`
     - `PFM_wall_r = 0.001`
     - `PFM_wall_offset = 0.25`
     - `PFM_wall_rho = 0.25`
     - `PFM_angle = 2.0944`
     - `PFM_l = 1.0`
     - `PFM_absSpeed = 0.5`
     - `PFM_targetdist = 0.5`
     - `PFM_vis = 0.0`
   - Save Bus before closing

5. Type into command window of MatLab:
• make_all_install
• make_motion_PFM

6. Open ‘motion_turtle.mdl’
• /home/robocup/svn/trunk/src/Turtle2/Motion/motion_turtle.mdl
• Navigate to /motion_turtle/strategy & control/actions
• From library browser add an S-Function and put ’determinedirection’ in the S-function name field.
• Press right mouse click and click on create mask, add the text of mask.txt
• Connect all corresponding inputs to the S-function
• Connect the output of the S-function to a ’GoTo’ (from library) and edit the tag to ’acceleration_app’, change tag visibility to global
• File, Save
• Navigate to: ’motion_turtle/strategy & control/control/trajectory planner/xy trajectory planner’
  – Delete connection from ’parking brake’ to ’axy’
  – Open ’Global2Local.mdl’ via file open in matlab and copy the contents to the xy trajectory planner
  – Connect output from Global2Local to ’axy’
  – Open terminator block from library and connect the output from parking brake to the terminator block.
  – Save

7. Build the software in the MatLab command window:
• make_motion_PFM
• build_all
• build_sim_all
C PFM in C-code

//versie 17-7-2013 - 16:05

#define S_FUNCTION_NAME determinedirection
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"
#include <math.h>
#include <float.h>
#include "global_par.h"
#include "generic_functions.h"
#include "bus.h"
#include ".//action_handler/Actions.h"

int counter = 0; //counter to prevent non-fixable error in the boundary obstacle loading
int counterpos = 0; //counter for target

typedef struct{
    double x;
    double xglobal;
    double y;
    double yglobal;
    double vx;
    double vy;
    double r;
}Obstacle;

typedef struct{
    double x;
    double y;
}Force;

typedef struct{
    double m; //mass of robot
    double r; //radius of robot
    double vx;
    double vy;
    double ax;
    double ay;
    double imxx; //inverse mass xx
    double imxy; //inverse mass xy
    double imyx; //inverse mass yx
    double imyy; //inverse mass yy
}Robot;
typedef struct{
    double x;
    double y;
    double xglobal;
    double yglobal;
    double vx;
    double vy;
    double ax;
    double ay;
} Position;

//set Obstacles returns the number of obstacles and sets array obst with the correct values
int getObstacles(Obstacle* obst, double* avoidables_pos_global, double* avoidables_pos_local, double* avoidables_vel_global, double* avoidables_vel_local, double* avoidables_radius, int jOBST, int iOBST)
{
    for (iOBST=0; iOBST<NAVOIDABLES_FOR_STRATEGY; ++iOBST)
    {
        if (avoidables_pos_vel_radius_global[iOBST*5+4]>0.01)
        {
            avoidables_pos_global[2*jOBST] = avoidables_pos_vel_radius_global[iOBST*5];
            avoidables_pos_global[2*jOBST+1] = avoidables_pos_vel_radius_global[iOBST*5+1];
            avoidables_vel_global[2*jOBST] = avoidables_pos_vel_radius_global[iOBST*5+2];
            avoidables_vel_global[2*jOBST+1] = avoidables_pos_vel_radius_global[iOBST*5+3];
            avoidables_radius[jOBST] = avoidables_pos_vel_radius_global[iOBST*5+4];
            jOBST++; /* Number of obstacles */
        }
    }

    //globale coordinaten in obstacle zetten
    //obstacles in obst zetten
    for(iOBST = 0; iOBST < jOBST; iOBST++){
        obst[iOBST].xglobal = avoidables_pos_global[2*iOBST];
        obst[iOBST].yglobal = avoidables_pos_global[2*iOBST+1];
    }

    global2local((double*)avoidables_pos_global,jOBST,(double*)cur_xyo,(double*)avoidables_pos_local);
    global2localVel((double*)avoidables_vel_global,jOBST,(double*)cur_xyo,(double*)avoidables_vel_local);

    //obstacles in obst zetten
    for(iOBST = 0; iOBST < jOBST; iOBST++){
        obst[iOBST].x = avoidables_pos_local[2*iOBST];
    }
obst[iOBST].y = avoidables_pos_local[2*iOBST+1];
obst[iOBST].vx = avoidables_vel_local[2*iOBST];
obst[iOBST].vy = avoidables_vel_local[2*iOBST+1];
obst[iOBST].r = avoidables_pos_vel_radius_global[iOBST*5+4];
}
return jOBST;
}

/*calculate attractive force \( Fa = \text{getAttractiveForce}(target, PFM_m, PFM_alpha_p) \)/ */
Force getAttractiveForce(Position target, Robot turtle, double PFM_m, double PFM_n, double PFM_alpha_p, double PFM_alpha_v, double amax){
Force F;
F.x=0.0;F.y=0.0;
Position unitVector;

double rho_t = 0.0; //rho_t is distance between the target and the turtle
rho_t = sqrt(target.x*target.x + target.y*target.y); //Pythagoras distance = sqrt(x^2+y^2)

// avoid dividing by zero
if(rho_t > 0){
unitVector.x = target.x / (rho_t + DBL_EPSILON);
unitVector.y = target.y / (rho_t + + DBL_EPSILON);
F.x = PFM_m*PFM_alpha_p*pow(rho_t,(PFM_m-1.0))*unitVector.x;
F.y = PFM_m*PFM_alpha_p*pow(rho_t,(PFM_m-1.0))*unitVector.y;
}

double v_rt = 0.0;
v_rt = sqrt(turtle.vx*turtle.vx + turtle.vy*turtle.vy);
if(v_rt>0){
    unitVector.x = -turtle.vx/v_rt;
    unitVector.y = -turtle.vy/v_rt;
    F.x = F.x + PFM_n*PFM_alpha_v*pow(v_rt,(PFM_n-1.0))*unitVector.x;
    F.y = F.y + PFM_n*PFM_alpha_v*pow(v_rt,(PFM_n-1.0))*unitVector.y;
}

F.x = F.x + DBL_EPSILON;
F.y = F.y + DBL_EPSILON;

return F;
}
//Fr = getRepulsiveForce((Obstacle*)obst, turtle, jOBST, PFM_etha, amax);
Force getRepulsiveForce(Obstacle* obst, Robot turtle, int jOBST, Force Fa, double PFM_etha, double PFM_rho_O, double amax, double* cur_xyo, double PFM_etha_wall, double PFM_wall_rho, double PFM_vis){
    Force F;
    F.x=0.0;F.y=0.0;

    //check if there are any obstacles
    if(jOBST>0){
        int iOBST = 0;
        //iterate through all obstacles and calculate repulsive force
        for(iOBST = 0; iOBST < jOBST; iOBST++){
            double l_rho_s = sqrt(obst[iOBST].x*obst[iOBST].x + obst[iOBST].y*obst[iOBST].y) + DBL_EPSILON;
            Position n_ro, n_rop;
            n_ro.x = obst[iOBST].x/(l_rho_s + DBL_EPSILON);
            n_ro.y = obst[iOBST].y/(l_rho_s + DBL_EPSILON);
            //perpendicular
            n_rop.x = -n_ro.y;
            n_rop.y =   n_ro.x;

            double v_ro, v_rop;
            v_ro = ((obst[iOBST].vx - turtle.vx + DBL_EPSILON)*n_ro.x + (obst[iOBST].vy - turtle.vy + DBL_EPSILON)*n_ro.y);
            v_rop = ((obst[iOBST].vx - turtle.vx + DBL_EPSILON)*n_rop.x + (obst[iOBST].vy - turtle.vy + DBL_EPSILON)*n_rop.y);
            double rho_m = (v_ro*v_ro)/(amax);

            double distance = l_rho_s + DBL_EPSILON;
            double radius = PFM_rho_O + obst[iOBST].r + rho_m + turtle.r + DBL_EPSILON;
            double radiuswall = PFM_wall_rho + obst[iOBST].r + rho_m + turtle.r + DBL_EPSILON;

            if(iOBST==jOBST-1){
                if(PFM_vis > 0.2){
                    drawCross(obst[iOBST].xglobal, obst[iOBST].yglobal,GETID);
                    drawCircle(obst[iOBST].xglobal, obst[iOBST].yglobal, radiuswall,0,GETID+iOBST);
                }
            }  
        }
    }

    //check if wall obstacle, iOBST==jOBST: yes calculate repulsive wall; no: do normal obstacle
    if(iOBST==jOBST-1){
        if(distance - obst[iOBST].r -turtle.r - rho_m< PFM_wall_rho && v_ro < 0){ //

    }
}

37
\begin{verbatim}
F.x = F.x - ((\text{pow}(PFM_etha,1)/(distance*distance + DBL_EPSILON))*(1.0 - v_ro/amax))*n_ro.x;
F.y = F.y - ((\text{pow}(PFM_etha,1)/(distance*distance + DBL_EPSILON))*(1.0 - v_ro/amax))*n_ro.y;
if(PFM_vis > 0.2){
drawCircleColor(DD_BLUE,obst[iOBST].xglobal, obst[iOBST].yglobal, radiuswall,1,GETID);
}
F.x = F.x + ((PFM_etha*v_ro*v_rop)/(DBL_EPSILON + l_rho_s*amax*pow(l_rho_s - obst[iOBST].r,2)))*n_rop.x;
F.y = F.y + ((PFM_etha*v_ro*v_rop)/(DBL_EPSILON + l_rho_s*amax*pow(l_rho_s - obst[iOBST].r,2)))*n_rop.y;
}
else{
if(distance - obst[iOBST].r - turtle.r - rho_m < PFM_rho_O && v_ro < 0){ //
F.x = F.x - ((\text{pow}(PFM_etha,1)/(distance*distance))*(1.0 - v_ro/amax))*n_ro.x;
F.y = F.y - ((\text{pow}(PFM_etha,1)/(distance*distance))*(1.0 - v_ro/amax))*n_ro.y;
if(PFM_vis > 0.2){
drawCircleColor(DD_BLUE,obst[iOBST].xglobal, obst[iOBST].yglobal, radius,1,GETID+iOBST);
}
F.x = F.x + ((PFM_etha*v_ro*v_rop)/(DBL_EPSILON + l_rho_s*amax*pow(l_rho_s - obst[iOBST].r,2)))*n_rop.x;
F.y = F.y + ((PFM_etha*v_ro*v_rop)/(DBL_EPSILON + l_rho_s*amax*pow(l_rho_s - obst[iOBST].r,2)))*n_rop.y;
}
}
}
F.x = F.x + DBL_EPSILON;
F.y = F.y + DBL_EPSILON;
return F;
\end{verbatim}
double control_force_max;  /* Maximum control force */
double VelDes;  /* Desired velocity */
rho_t = sqrt(target.x*target.x + target.y*target.y);  //Pythagoras: distance = sqrt(x^2+y^2)

VelDes = dmin(vmax,sqrt(2*amax*rho_t));
Position VelError;
VelError.vx = VelDes*N.x - cur_xydot_local[0];
VelError.vy = VelDes*N.y - cur_xydot_local[1];

control_force_max = turtle.m*amax;
Force CF;
CF.x = control_force_max*(VelError.vx/dmax(sqrt(VelError.vx*VelError.vx+VelError.vy*VelError.vy),DBL_EPSILON));
CF.y = control_force_max*(VelError.vy/dmax(sqrt(VelError.vx*VelError.vx+VelError.vy*VelError.vy),DBL_EPSILON));

turtle.ax = turtle.imxx*(CF.x-N.x) + turtle.imxy*(CF.y - N.y);
turtle.ay = turtle.imyx*(CF.x-N.x) + turtle.imyy*(CF.y - N.y);

return turtle;

}  //end of function "turtlecontrol"
Position unitVector;
unitVector.x = speed.vx / sqrt(speed.vx*speed.vx + speed.vy*speed.vy);
unitVector.y = speed.vy / sqrt(speed.vx*speed.vx + speed.vy*speed.vy);

//determine labda's:
double labda[4];
double maxlabda = 0.0;
//labda[0] for b[0].x
labda[0] = (b[0].x - cur_xyo[0])/unitVector.x;
//calculate b[0].y
b[0].y = cur_xyo[1] + labda[0]*unitVector.y;
if(labda[0] > maxlabda){maxlabda = labda[0];}

labda[1] = (b[1].x - cur_xyo[0])/unitVector.x;
//calculate b[1].y
b[1].y = cur_xyo[1] + labda[1]*unitVector.y;
if(labda[1] > maxlabda){maxlabda = labda[1];}

labda[2] = (b[2].y - cur_xyo[1])/unitVector.y;
//calculate b[2].x
b[2].x = cur_xyo[0] + labda[2]*unitVector.x;
if(labda[2] > maxlabda){maxlabda = labda[2];}

labda[3] = (b[3].y - cur_xyo[1])/unitVector.y;
//calculate b[3].x
b[3].x = cur_xyo[0] + labda[3]*unitVector.x;
if(labda[3] > maxlabda){maxlabda = labda[3];}

//find minimum positive labda
int i = 0;
int minpos = 0;
for(i = 0; i<4; i++){
    if(labda[i] > 0.0 && labda[i] < maxlabda){
        maxlabda = labda[i];
        minpos = i;
    }
}

double globalbp[2];
globalbp[0] = b[minpos].x;
globalbp[1] = b[minpos].y;
bound.xglobal = b[minpos].x;
bound.yglobal = b[minpos].y;
double localbp[2];
global2local((double*)globalbp, 1, (double*)cur_xyo, (double*)localbp);

bound.x = localbp[0];
bound.y = localbp[1];
drawCircle(globalbp[0], globalbp[1], PFM_wall_r, 1, GETID);

return bound;
}

Force getLocalMinimumForce(Obstacle* obst, Robot turtle, Position target, int jOBST, Force Fn, Force Fa, Force Fr, double ... cur_xydot, double* cur_xyo, double PFM_angle, double PFM_b, double PFM_absSpeed, double PFM_targetdist, double PFM_vis){

Force Fescape;
Fescape.x = 0.0;
Fescape.y = 0.0;

//calculate angle between Fattr and Frep

double absFa = sqrt(Fa.x*Fa.x + Fa.y*Fa.y);
double absFr = sqrt(Fr.x*Fr.x + Fr.y*Fr.y);

//Check for local minimum and apply the correct force
double b = sqrt(Fn.x*Fn.x + Fn.y*Fn.y)/sqrt(Fr.x*Fr.x + Fr.y*Fr.y);
double dotFaFr = Fa.x*Fr.x + Fa.y*Fr.y;
double absSpeed = cur_xydot[0]*cur_xydot[0] + cur_xydot[1]*cur_xydot[1];

//check if one of the forces is not equal to zero
if (!(absFa <= 0.00001 || absFr <= 0.00001)){
    double pi; pi = 3.14159265;
    double costheta;
    costheta = (dotFaFr)/(absFa*absFr);
    double angleradian;
    double angledegrees;
    angleradian = acos(costheta);
    angledegrees = angleradian * (180/pi);

    int iOBST = 0;

    //check for local minima and if robot is NOT near target position
    double distotarget = sqrt(target.x*target.x + target.y*target.y);
    if (angleradian > PFM_angle && b<PFM_b && absSpeed < PFM_absSpeed && distotarget > PFM_targetdist){
        //local minima detected and NOT near target position

        //Find obstacle which is closest to apply escape force
double rhoinf = 1000.0;
int posminimum = -1;
for(iOBST = 0; iOBST < jOBST; iOBST++){
    //iterate through all obstacles and find out which one is closest
    double l_rho_s = sqrt(obst[iOBST].x*obst[iOBST].x + obst[iOBST].y*obst[iOBST].y) + DBL_EPSILON;
    if(l_rho_s<rhoinf){
        rhoinf = l_rho_s;
        posminimum = iOBST;
    }
}

//Calculate perpendicular unit direction of vector Frep
Position unitVector, Position unitVectorP; //perpendicular
unitVector.x = Fr.x/absFr;
unitVector.y = Fr.y/absFr;
unitVectorP.x = -unitVector.y;
unitVectorP.y = unitVector.x;

//Check angle between unitVectorP and Vrobot
if(acos((turtle.vx*unitVectorP.x + turtle.vy*unitVectorP.y)/(sqrt(turtle.vx*turtle.vx + turtle.vy*turtle.vy)*sqrt(unitVectorP.x*unitVectorP.x + unitVectorP.y*unitVectorP.y)))> 0.5*pi){
    unitVectorP.x = -unitVectorP.x;
    unitVectorP.y = -unitVectorP.y;
}

//Order of attractive force
double z = log(sqrt(Fr.x*Fr.x + Fr.y*Fr.y))/log(10);

Position n_ro, n_rop;
n_ro.x = obst[posminimum].x/(rhoinf + DBL_EPSILON);
n_ro.y = obst[posminimum].y/(rhoinf + DBL_EPSILON);
//perpendicular
n_rop.x = -n_ro.y;
n_rop.y = n_ro.x;
double v_ro, v_rop;
v_ro = ((obst[posminimum].vx - turtle.vx + DBL_EPSILON)*n_ro.x + (obst[posminimum].vy - turtle.vy + DBL_EPSILON)*n_ro.y);
v_rop = ((obst[posminimum].vx - turtle.vx + DBL_EPSILON)*n_rop.x + (obst[posminimum].vy - turtle.vy + DBL_EPSILON)*n_rop.y);
double rho_m = (v_ro*v_ro)/(amax);

Fescape.x = pow(10.0,z)*(1/pow((rhoinf-turtle.r-rho_m),2))*unitVectorP.x;
Fescape.y = pow(10.0,z)*(1/pow((rhoinf-turtle.r-rho_m),2))*unitVectorP.y;
/draw
double perp[2];
perp[0] = Fescape.x;
perp[1] = Fescape.y;
local2globalVel((double*)perp,1,(double*)cur_xyo,(double*)perp);
if(PFM_vis > 0.2){
  vector_t vRpos; vRpos.x = cur_xyo[0]; vRpos.y = cur_xyo[1];
  vector_t vFp; vFp.x = vRpos.x + perp[0]; vFp.y = vRpos.y + perp[1];
  drawLineVectorsFancy(DD_GREEN, vRpos, vFp,0.1,-1,GETID);
}

return Fescape;

}/**-----------------------------*/
/*--------- S-function methods -------------*/
static void mdlInitializeSizes(SimStruct *S)
{
  /*PARAMETERS*/
  ssSetNumSFcnParams(S, 0);
  if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
    return; /* Parameter mismatch will be reported by Simulink */
  }

  /*INPUT PORTS*/
  if (!ssSetNumInputPorts(S, 7)) return;

  ssSetInputPortWidth(S, 0, 2);
  ssSetInputPortDataType(S, 0, SS_DOUBLE);
  ssSetInputPortDirectFeedThrough(S, 0, 1);
  ssSetInputPortRequiredContiguous(S, 0, 1);

  ssSetInputPortWidth(S, 1, 5*NAVOIDABLES_FOR_STRATEGY);
  ssSetInputPortDataType(S, 1, SS_DOUBLE);
  ssSetInputPortDirectFeedThrough(S, 1, 1);
  ssSetInputPortRequiredContiguous(S, 1, 1);

  ssSetInputPortWidth(S, 2, SENSORFUSIONBUS_SIZE);
ssSetInputPortDataType(S, 2, SS_INT8);
ssSetInputPortDirectFeedThrough(S, 2, 1);
ssSetInputPortRequiredContiguous(S, 2, 1);
ssSetInputPortWidth(S, 3, CONTROLBUS_SIZE);
ssSetInputPortDataType(S, 3, SS_INT8);
ssSetInputPortDirectFeedThrough(S, 3, 1);
ssSetInputPortRequiredContiguous(S, 3, 1);
ssSetInputPortWidth(S, 4, TUNABLE_PARDATA_MOTION_BUS_SIZE);
ssSetInputPortDataType(S, 4, SS_INT8);
ssSetInputPortDirectFeedThrough(S, 4, 1);
ssSetInputPortRequiredContiguous(S, 4, 1);
ssSetInputPortWidth(S, 5, 1);
ssSetInputPortDataType(S, 5, SS_DOUBLE);
ssSetInputPortDirectFeedThrough(S, 5, 1);
ssSetInputPortRequiredContiguous(S, 5, 1);
ssSetInputPortWidth(S, 6, 1);
ssSetInputPortDataType(S, 6, SS_DOUBLE);
ssSetInputPortDirectFeedThrough(S, 6, 1);
ssSetInputPortRequiredContiguous(S, 6, 1);

/*OUTPUT PORTS*/
if (!ssSetNumOutputPorts(S, 3)) return;

ssSetOutputPortWidth(S, 0, 2);
ssSetOutputPortDataType(S, 0, SS_DOUBLE);

ssSetOutputPortWidth(S, 1, 2);
ssSetOutputPortDataType(S, 1, SS_DOUBLE);

ssSetOutputPortWidth(S, 2, 2);
ssSetOutputPortDataType(S, 2, SS_DOUBLE);

ssSetNumIWork(S, 1);

}

static void mdlInitializeSampleTimes(SimStruct *S)
ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
ssSetOffsetTime(S, 0, 0.0);

#define MDL_INITIALIZE_CONDITIONS
#if defined(MDL_INITIALIZE_CONDITIONS)
static void mdlInitializeConditions(SimStruct *S)
{
}
#endif /* INITIALIZE_CONDITIONS */

#define MDL_START /* Change to #undef to remove function */
#if defined(MDL_START)
static void mdlStart(SimStruct *S)
{
}
#endif /* MDL_START */

static void mdlOutputs(SimStruct *S, int_T tid)
{

/* get pointers to inputs */
const real_T *target_global = (const real_T*) ssGetInputPortSignal(S,0);
const real_T *avoidables_pos_vel_radius_global = (const real_T*) ssGetInputPortSignal(S,1); /* [x, y, xd, yd, radius] */
psensorfusionbus_t psensorfusionbus = (psensorfusionbus_t) ssGetInputPortRealSignalPtrs(S,2); /* sensorfusionbus */
pcontrolbus_t pcontrolbus = (pcontrolbus_t) ssGetInputPortRealSignalPtrs(S,3); /* controlbus */
const ptunable_pardata_motion_bus_t ptunable_par_motion = (ptunable_pardata_motion_bus_t) ssGetInputPortRealSignalPtrs(S,4); /* motion tuning bus */
const real_T* refboxVel = (const real_T*) ssGetInputPortSignal(S,5);
const real_T* refboxAcc = (const real_T*) ssGetInputPortSignal(S,6);

real_T *cur_xyo = psensorfusionbus->cur_xyo;   /* Current position global */
real_T *motion_offset = psensorfusionbus->motion_offset;
real_T *cur_xydot = pcontrolbus->cur_xydot;
real_T vmax = 0.0, amax = 0.0;

vmax = dmin(ptunable_par_motion->SF_v_max_move,*refboxVel);
amax = dmin(ptunable_par_motion->SF_a_max_move,*refboxAcc);
real_T PFM_m = ptunable_par_motion->PFM_m; /* Positive constant - Position */
real_T PFM_n = ptunable_par_motion->PFM_n; /* Positive constant - velocity */
real_T PFM_alpha_p = ptunable_par_motion->PFM_alpha_p; /* Positive scaling factor for attractive function - Position */
real_T PFM_alpha_v = ptunable_par_motion->PFM_alpha_v; /* Positive scaling factor for attractive function - Velocity */
real_T PFM_rho_O = ptunable_par_motion->PFM_rho_O; /* Influence radius of obstacle */
real_T PFM_etha = ptunable_par_motion->PFM_etha; /* Positive scaling factor for repulsive function */
real_T PFM_etha_wall = ptunable_par_motion->PFM_etha_wall; /* Positive scaling factor for repulsive wall function */
real_T PFM_wall_r = ptunable_par_motion->PFM_wall_r; /* Radius of wall obstacle */
real_T PFM_wall_offset = ptunable_par_motion->PFM_wall_offset; /* Line offset of wall obstacle */
real_T PFM_wall_rho = ptunable_par_motion->PFM_wall_rho; /* Influence radius of wall obstacle */
real_T PFM_angle = ptunable_par_motion->PFM_angle; /* Angle for local minimum detection */
real_T PFM_b = ptunable_par_motion->PFM_b; /* Ratio of \|F_{total}\|/\|F_{rep}\| for local minimum detection */
real_T PFM_absSpeed = ptunable_par_motion->PFM_absSpeed; /* Maximum velocity of robot for local minumum detection */
real_T PFM_targetdist = ptunable_par_motion->PFM_targetdist; /* Distance robot to target where the escape force does not apply */
real_T PFM_vis = ptunable_par_motion->PFM_vis; /* If greater as 0.2 visualization in the simulator is on */

/*get pointers to outputs*/
double *acceleration_app = (double*)ssGetOutputPortRealSignal(S,0);
double *Fattractive = (double*)ssGetOutputPortRealSignal(S,1);
double *Frepulsive = (double*)ssGetOutputPortRealSignal(S,2);

/*declarations*/
double avoidables_pos_global[2*NAVOIDABLES_FOR_STRATEGY]; /* Obstacle position global */
double avoidables_pos_local[2*NAVOIDABLES_FOR_STRATEGY]; /* Obstacle position local */
double avoidables_vel_global[2*NAVOIDABLES_FOR_STRATEGY]; /* Obstacle velocity global */
double avoidables_vel_local[2*NAVOIDABLES_FOR_STRATEGY]; /* Obstacle velocity local */
double avoidables_radius[NAVOIDABLES_FOR_STRATEGY]; /* Obstacle radius */
double desired_force[2]; /* Force in the desired direction */
double cur_xydot_global[2]; /* robot velocity - global */
double cur_xydot_local[2]; /* robot velocity - local */
double control_force[2]; /* Applied control force */
double Ndirection[2]; /* Desired direction of motion */
double acceleration_app_local[2]; /* Applied acceleration */
double target_local[2]; /* Target position */

daether, cur_xydot, control, target, Ndirection, acceleration_app, target
/* Initial values */
target_local[0] = 0.0;
target_local[1] = 0.0;
desired_force[0] = 0.0;
desired_force[1] = 0.0;
avoidables_pos_global[0] = 0.0;
avoidables_pos_global[1] = 0.0;
avoidables_vel_global[0] = 0.0;
avoidables_vel_global[1] = 0.0;
avoidables_radius[0] = 0.0;
acceleration_app_local[0] = 0.0;
acceleration_app_local[1] = 0.0;

/*global to local*/
local2globalVel((double*)cur_xydot,1,(double*)motion_offset,cur_xydot_global);
global2localVel((double*)cur_xydot_global,1,(double*)cur_xyo,cur_xydot_local);

/* Initialize robot */
Robot turtle;
turtle.m = 35.0;  //mass of robot
turtle.r = 0.25;  //radius of robot
turtle.imxx = 0.0286;
turtle.imxy = 0.0;
turtle.imyx = 0.0;
turtle.imyy = 0.0286;
turtle.vx = cur_xydot_local[0];
turtle.vy = cur_xydot_local[1];
turtle.ax = 0.0;
turtle.ay = 0.0;

/* Initialize attractive force and set it to zero */
Force Fa;
Fa.x = 0.0;
Fa.y = 0.0;

/* Initialize repulsive force and set it to zero */
Force Fr;
Fr.x = 0.0;
Fr.y = 0.0;

/* Initialize net force and set it to zero */
Force Fn;
Fn.x = 0.0;
Fn.y = 0.0;

/* Initialize escape force and set it to zero*/
Force Fe;
Fe.x = 0.0;
Fe.y = 0.0;

/* Initialize target and convert to local position */
//let the robot switch between targets
//check if robot and target are within 0.5m then goto next target
Position targets[3];
targets[0].x = 0.0; targets[0].y = 0.0;
targets[1].x = -3.75; targets[1].y = 5.75;
targets[2].x = -3.75; targets[2].y = -5.75;

double ctarget[2];
ctarget[0] = targets[counterpos].x; ctarget[1] = targets[counterpos].y;
if(PFM_vis > 0.2){
drawCross(ctarget[0], ctarget[1], GETID);
}
//check if near target if yes goto next target
if(sqrt(pow(ctarget[0]-cur_xyo[0],2) + pow(ctarget[1]-cur_xyo[1],2))<0.5){
counterpos = counterpos+1;
counterpos = counterpos%3;
}

global2local((double*)ctarget,1,(double*)cur_xyo,target_local);
Position target;
target.x = target_local[0];
target.y = target_local[1];

/* Load obstacle positions and radius */

int iOBST = 0, jOBST = 0; //Where jOBST is the total number of obstacles
Obstacle obst[NAVOIDABLES_FOR_STRATEGY]; //Create array of obstacles

//returns the number of obstacles and set local coordinates to obstacles array
//obst[0 ... jOBST-1] contains all the obstacles, obst[0].x holds local x position
jOBST = getObstacles((Obstacle*)obst, (double*)avoidables_pos_global,(double*)avoidables_pos_local,(double*)avoidables_radius,(double*)cur_xyo, jOBST, iOBST);

//Add boundary obstacle
if(counter >= 10000){
    //use counter to avoid unknown error
Position boundaryposition;
//boundaryObstacle returns the local x and y position of the boundary obstacle
boundaryposition = boundaryObstacle((double*)cur_xydot, (double*)cur_xyo, PFM_w,
obst[jOBST].x = boundaryposition.x; obst[jOBST].y = boundaryposition.y;
obst[jOBST].xglobal = boundaryposition.xglobal; obst[jOBST].yglobal = boundaryposition.yglobal;
obst[jOBST].vx = 0.0; obst[jOBST].vy = 0.0; obst[jOBST].r = PFM_wall_r;
jOBST++;
counter = 10001;
}
counter++;

//get attractive force
Fa = getAttractiveForce(target, turtle, PFM_m, PFM_n, PFM_alpha_p, PFM_alpha_v, amax);

//get repulsive force
Fr = getRepulsiveForce((Obstacle*)obst, turtle, jOBST, Fa, PFM_etha, PFM_rho_O, amax, (double*)cur_xyo, PFM_etha_wall, PFM_wall_rho, PFM_vis);

//Add attractive + repulsive force
Fn.x = Fa.x + Fr.x;
Fn.y = Fa.y + Fr.y;

//Get escape force
Fe = getLocalMinimumForce((Obstacle*) obst, turtle, target, jOBST, Fn, Fa, Fr, amax,
Fn.x = Fn.x + Fe.x;
Fn.y = Fn.y + Fe.y;

//calculate the acceleration of the turtle
turtle = applyInertia(turtle, Fn, target, vmax, amax,(double*)cur_xydot_local);

acceleration_app_local[0] = turtle.ax;
acceleration_app_local[1] = turtle.ay;
/* Local to global */

double Fatt[2];
Fatt[0] = Fa.x;
Fatt[1] = Fa.y;
double Frep_total[2];
Frep_total[0] = Fr.x;
Frep_total[1] = Fr.y;
local2globalVel((double*)Fatt,1,(double*)cur_xyo,(double*)Fattractive);
local2globalVel((double*)Frep_total,1,(double*)cur_xyo,(double*)Frepulsive);
//draw force vectors
if(PFM_vis > 0.2)
{
    vector_t vRpos; vRpos.x = cur_xyo[0]; vRpos.y = cur_xyo[1];
    vector_t vFa; vFa.x = Fattractive[0] + vRpos.x; vFa.y = Fattractive[1] + vRpos.y;
    vector_t vFr; vFr.x = Frepulsive[0] + vRpos.x; vFr.y = Frepulsive[1] + vRpos.y;
    vector_t vFn; vFn.x = Fattractive[0] + Frepulsive[0] + vRpos.x; vFn.y = Fattractive[1] + Frepulsive[1] + vRpos.y;

    drawLineVectorsFancy(DD_BLACK, vRpos, vFa, 0.1,-1,GETID);
    drawLineVectorsFancy(DD_RED, vRpos, vFr, 0.1,-1,GETID);
    drawLineVectorsFancy(DD_YELLOW, vRpos, vFn, 0.1,-1,GETID);
}

local2globalVel((double*)acceleration_app_local,1,(double*)cur_xyo,(double*)acceleration_app);

static void mdlTerminate(SimStruct *S)
{
}

#ifdef MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"       /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"        /* Code generation registration function */
#endif
Title : Motion planning for Tech United
Assignment : BEP, 2 students, dual assignment
Supervisor : Dr.ir. René van de Molengraft m.j.g.v.d.molengraft@tue.nl
Coach(es) : Dr.ir. Gerrit Naus g.j.l.naus@tue.nl
Bas Coenen (daily coaching) s.a.m.coenen@student.tue.nl
Group : Control Systems Technology group prof.dr.ir. M. Steinbuch
Starting date : 22/04/13 End date :

Introduction
RoboCup is an international research and education initiative focusing on artificial intelligence (AI) and intelligent robotics. To stimulate participation, a competition is set up, including rescue leagues, service robotic leagues and soccer leagues in different forms. The highly dynamic environment of soccer is a perfect platform for application of AI and intelligent robotics and, as such, leads to innovations that benefit society [1]. Tech United Eindhoven participates in three leagues: the @Home, the Humanoid, and the Middle-Size League [2]. In the Middle-Size League, a team of five robots (called TURTLEs) play soccer autonomously. These wheeled robots are 80 cm in height, weigh 35 kg, reach a top speed up to 5 m/s and shoot a ball with a speed of 10 m/s. The robots are required to have all sensors on-board. The fast-paced soccer game of this league poses great challenges in several fields such as mechatronics, software, strategy, cooperation and AI [2].

Problem definition
An important part of the TURTLE control software involves motion planning. Motion planning includes high-level decisions on, e.g., where to position on the field or via what side to attack, and low-level computations on how to avoid collisions with opponents and on the exact trajectory to drive. Given the currently used motion planner, various points for improvement have been identified. One important example involves taking into account velocity estimation and prediction of opponents. This is, however, difficult or even impossible to do using the currently implemented planner. Based on a literature study [3], the potential field method (PFM) has been identified as a suitable method including the proposed improvements. Additional research and implementation are required to evaluate this method in practice.

Assignment / goals
The goal of this assignment is twofold. On the one hand, the opponent velocity estimation and prediction need evaluation and possibly improvement. On the other hand, further evaluation of the potential field method requires implementation and testing of the method in the TURTLE software.

As discussed in the problem definition, the estimation and the prediction of the velocity of opponents is an important aspect in improving the motion planner. An opponent velocity estimation algorithm is already available and implemented in the software. However, evaluation of this algorithm requires further research: i) what is the accuracy of the current estimations and predictions, ii) is this accuracy sufficient to use it in a motion planner and to make the planner robust against fast-moving obstacles, iii) what are the limitations of the current estimations, and iv) can the accuracy be improved. Data from previous matches during championships is available for analysis. Furthermore, new data can easily be generated by testing with the robots.

Previous BEP assignments have resulted in a potential field method for the TURTLEs, that has been implemented and evaluated in i) a MATLAB simulation environment [4] and ii) the TURTLE software simulation environment [5]. Several practical issues have already been addressed, however, further
evaluation of the method requires implementation on a real robot and on-field testing. The TURTLE software is written in C, within a MATLAB/Simulink environment.

The two parts of the assignment can be carried out separately, however, cooperation is highly recommendable to combine both the theoretical and practical aspects of the assignment.

Further information